1 OBJECT-ORIENTED SCIENTIFIC PROGRAMMING WITH FORTRAN 90

Charles D. Norton*

National Research Council, U.S.A
E-Mail: Charles.D.Norton@jpl.nasa.gov

Abstract: Fortran 90 is a modern language that introduces many important new features beneficial for scientific programming. While the array-syntax notation has received the most attention, we have found that many modern software development techniques can be supported by this language, including object-oriented concepts.

While Fortran 90 is not a full object-oriented language it can directly support many of the important features of such languages. Features not directly supported can be emulated by software constructs. It is backward compatible with Fortran 77, and a subset of HPF, so new concepts can be introduced into existing programs in a controlled manner. This allows experienced Fortran 77 programmers to modernize their software, making it easier to understand, modify, share, explain, and extend.

We discuss our experiences in plasma particle simulation and unstructured adaptive mesh refinement on supercomputers, illustrating the features of Fortran 90 that support the object-oriented methodology. All of our Fortran 90 programs achieve high performance with the benefits of modern abstraction modeling capabilities.

*Currently in residence at the National Aeronautical and Space Administration's Jet Propulsion Laboratory, California Institute of Technology, U.S.A.

Keywords: Fortran 90, Supercomputing, Object-Oriented, Adaptive Mesh Refinement, Scientific Programming, Plasma Physics.

1.1 INTRODUCTION

Scientific application programming involves unifying abstract physical concepts and numerical models with sophisticated programming techniques that require patience and experience to master. Furthermore, codes typically written by scientists are constantly changing to model new physical effects. These factors can contribute to long development periods, unexpected errors, and software that is difficult to comprehend, particularly when multiple developers are involved.

The Fortran 90 programming language (Ellis et al., 1994) addresses the needs of modern scientific programming by providing features that raise the level of abstraction, without sacrificing performance. Consider a 3D parallel plasma particle-in-cell (PIC) program in Fortran 77 which will typically define the particles, charge density field, force field, and routines to push particles and deposit charge. This is a segment of the main program where many details have been omitted.

```
dimension part(idimp, npmax), q(nx, ny, nzpmx)
dimension fx(nx, ny, nzpmx), fy(nx, ny, nzpmx), fz(nx, ny, nzpmx)
data qme, dt /-1.,.2/
call push(part,fx,fy,fz,npp,qtme,dt,wke,nx,ny,npmax,nzpmx,...)
call dpost(part,q,npp,noff,qme,nx,ny,idimp,npmax,nzpmx)
```

Note that the arrays must be dimensioned at compile-time. Also parameters must either be passed by reference, creating long argument lists, or kept in common and exposed to inadvertent modification. Such an organization is complex to maintain, especially as codes are modified for new experiments.

Using the new features of Fortran 90, abstractions can be introduced that clarify the organization of the code. The Fortran 90 version is more readable while designed for modification and extension.

```
use partition_module ; use plasma_module
type (species) :: electrons
type (scalarfield) :: charge_density
type (vectorfield) :: efield
type (slabpartition) :: edges
real :: dt = .2
call plasma_particle_push( electrons, efield, edges, dt )
call plasma_deposit_charge( electrons, charge_density, edges )
```

This style of object-oriented programming, where the basic data unit is an "object" that shields its internal data from misuse by providing public routines to manipulate it, allows such a code to be designed and written. Object-Oriented programming clarifies software while increasing safety and communi-

cation among developers, but its benefits are only useful for sufficiently large and complex programs.

While Fortran 90 is not an object-oriented language, the new features allow most of these concepts to be modeled directly. (Some concepts are more complex to emulate.) In the following, we will describe how object-oriented concepts can be modeled in Fortran 90, the application of these ideas to plasma PIC programming on supercomputers, parallel unstructured adaptive mesh refinement, and the future of Fortran programming (represented by Fortran 2000) that will contain explicit object-oriented features.

1.2 MODELING OBJECT-ORIENTED CONCEPTS IN FORTRAN 90

Object-Oriented programming (OOP) has received wide acceptance, and great interest, throughout the computational science community as an attractive approach to address the needs of modern simulation. Proper use of OOP ensures that programs can be written safely, since the internal implementation details of the data objects are hidden. This allows the internal structure of objects and their operations to be modified (to improve efficiency perhaps), but the overall structure of the code using the objects can remain unchanged. In other words, objects are an encapsulation of data and routines.

These objects represent abstractions. Another important concept is the notion of inheritance, which allows new abstractions to be created by preserving features of existing abstractions. This allows objects to gain new features through some form of code reuse. Additionally, polymorphism allows routines to be applied to a variety of objects that share some relationship, but the specific action taken varies dynamically based on the object's type. These ideas are mechanisms for writing applications that more closely represent the problem at hand. As a result, a number of programming languages support OOP concepts in some manner.

Fortran 90 is well-known for introducing array-syntax operations and dynamic memory management. While useful, this represents a small subset of the powerful new features available for scientific programming. Fortran 90 is backward compatible with Fortran 77 and, since it is a subset of High Performance Fortran (HPF), it provides a migration path for data-parallel programming. Fortran 90 type-checks parameters to routines, so passing the wrong arguments to a function will generate a compile-time error. Additionally, the automatic creation of implicit variables can be suppressed reducing unexpected results.

However, more powerful features include derived-types, which allow user-defined types to be created from existing intrinsic types and previously defined derived-types. Many forms of dynamic memory management operations are now available, including dynamic arrays and pointers. These new Fortran 90 constructs are objects that know information such as their size, whether they have been allocated, and if they refer to valid data. Fortran 90 modules allow routines to be associated with types and data defined within the module. These modules can be used in various ways, to bring new functionality to program

units. Components of the module can be private and/or public allowing interfaces to be constructed that control the accessibility of module components. Additionally, operator and routine overloading are supported (name reuse), allowing the proper routine to be called automatically based on the number and types of the arguments. Optional arguments are supported, as well as generic procedures that allow a single routine name to be used while the action taken differs based on the type of the parameter. All of these features can be used to support an object-oriented programming methodology (Decyk et al., 1997a).

1.3 APPLICATION: PLASMA PIC PROGRAMMING ON SUPERCOMPUTERS

Computer simulations are very useful for understanding and predicting the transport of particles and energy in fusion energy devices called tokamaks (Birdsall and Langdon, 1991). Tokamaks, which are toroidal in shape, confine the plasma with a combination of an external toroidal magnetic field and a self-generated poloidal magnetic field. Understanding plasma confinement in tokamaks could lead to the practical development of fusion energy as an alternative fuel source—unfortunately confinement is not well understood and is worse than desired.

PIC codes integrate the trajectories of many particles subject to electromagnetic forces, both external and self-generated. The General Concurrent PIC Algorithm (Liewer and Decyk, 1989), which partitions particles and fields among the processors of a distributed-memory supercomputer, can be programmed using a single-program multiple-data (SPMD) design approach. Although the Fortran 77 versions of these programs have been well-benchmarked and are scalable with nearly 100% efficiency, there is an increasing interest within the scientific community to apply object-oriented principles to enhance new code development.

In the introduction, we illustrated how Fortran 77 features could be modeled using Fortran 90 constructs. In designing the PIC programs, basic constructs like particles (individually and collectively), fields (scalar and vector, real and complex), distribution operations, diagnostics, and partitioning schemes were created as abstractions using Fortran 90 modules. Fortran 90 objects are defined by derived types within modules where the public routines that operate on these objects are visible whenever the object is "used". (The private components of the module are only accessible within module defined routines.)

A portion of the species module (Figure 1.1) illustrates how data and routines can be encapsulated using object-oriented concepts. This module defines the particle collection, where the interface to the particle Maxwellian distribution routine is included.

Some OOP concepts, such as inheritance, had limited usefulness while runtime polymorphism was used infrequently. Our experience has shown that these features, while sometimes appropriate for general purpose programming, do not seem to be as useful in scientific programming. Well-defined interfaces, that support manipulation of abstractions, were more important. More details on

Figure 1.1 Abstract of Fortran 90 module for particle species.

```
module species_module
    use distribution_module ; use partition_module
    implicit none
    type particle
        private
        real :: x, y, z, vx, vy, vz
    end type particle
    type species
        real :: qm, qbm, ek
        integer :: nop, npp
        type (particle), dimension(:), pointer :: p
    end type species
contains
    subroutine species_distribution(this, edges, distf)
        type (species), intent (out) :: this
        type (slabpartition), intent (in) :: edges
        type (distfcn), intent (in) :: distf
            ! subroutine body
    end subroutine species_distribution
    ! additional member routines...
end module species_module
```

the overall structure of the code can be found in (Norton et al., 1995; Norton et al., 1997).

The wall-clock execution times for the 3D parallel PIC code written in Fortran 90, Fortran 77, and C++ are illustrated in Table 1.1. Although our experience has been that Fortran 90 continually outperforms C++ on complete programs, generally by a factor of two, others have performance results that indicate that C++ can sometimes outperform Fortran 90 on some computational kernels (Cary et al., 1997). (In these cases, "expression templates" are introduced as a compile-time optimization to speed up complicated array operations.)

Table 1.1 3D Parallel Plasma PIC Experiments on the Cornell Theory Center IBM SP2 (32 Processors, 8M Particles, 260K Grid Points).

Language	*Compiler*	*P2SC Super Chips*	*P2SC Optimized*
Fortran 90	IBM xlf90	622.60s	488.88s
Fortran 77	IBM xlf	668.03s	537.95s
C++	KAI KCC	1316.20s	1173.31s

The most aggressive compiler options produced the fastest timings, seen in Table 1.1. The KAI C++ compiler with +K3 -O3 -abstract_pointer spent over 2 hours in the compilation process. The IBM F90 compiler with -O3 -qlanglvl=90std -qstrict -qalias=noaryovrlp used 5 minutes for compilation. (The KAI compiler is generally considered the most efficient C++ compiler when objects are used. This compiler generated slightly faster executables than the IBM C++ compiler.) Applying the hardware optimization switches -qarch=pwr2 -qtune=pwr2 introduced additional performance improvements specific to the P2SC processors.

We have found Fortran 90 very useful, and generally safer with higher performance than C++ and sometimes Fortran 77, for large problems on supercomputers. Fortran 90 derived-type objects improved cache utilization, for large problems, over Fortran 77. (The C++ and Fortran 90 objects had the same storage organization.) Fortran 90 is less powerful than C++, since it has fewer features and those available can be restricted to enhance performance, but many of the advanced features of C++ have not been required in scientific computing. Nevertheless, advanced C++ features may be more appropriate for other problem domains (Decyk et al., 1997b; Norton et al., 1997).

1.4 APPLICATION: PARALLEL UNSTRUCTURED ADAPTIVE MESH REFINEMENT

Adaptive mesh refinement is an advanced numerical technique very useful in solving partial differential equations. Essentially, adaptive techniques utilize a descretized computational domain that is subsequently refined/coarsened in areas where additional resolution is required. Parallel approaches are necessary for large problems, but the implementation strategies can be complex unless good design techniques are applied.

One of the major benefits of Fortran 90 is that codes can be structured using the principles of object-oriented programming. This allows the development of a parallel adaptive mesh refinement (PAMR) code where interfaces can be defined in terms of mesh components, yet the internal implementation details are hidden. These principles also simplify handling interlanguage communication, sometimes necessary when additional packages are interfaced to new codes. Using Fortran 90's abstraction techniques, for example, a mesh can be loaded into the system, distributed across the processors, the PAMR internal data structure can be created, and the mesh can be repartitioned and migrated to new processors (all in parallel) with a few simple statements as shown in Figure 1.2.

A user could link in the routines that support parallel adaptive mesh refinement then, as long as the data format from the mesh generation package conforms to one of the specified formats, the capabilities required for PAMR are available. We now describe the Fortran 90 implementation details that make this possible.

Figure 1.2 A main program with selected PAMR library calls.

```
program pamr
use mpi_module ; use mesh_module ; use misc_module
implicit none
integer :: ierror
character(len=8) :: input_mesh_file
type (mesh) :: in_mesh
call MPI_INIT( ierror )
      input_mesh_file = mesh_name( iam )
      call mesh_create_incore( in_mesh, input_mesh_file )
      call mesh_repartition( in_mesh )
      call mesh_visualize( in_mesh, "visfile.plt" )
call MPI_FINALIZE( ierror )
end program pamr
```

1.4.1 Basic Mesh Definition

Fortran 90 modules allow data types to be defined in combination with related routines. In our system the mesh is described, in part, as shown in Figure 1.3. In two-dimensions, the mesh is a Fortran 90 object containing nodes, edges, elements, and reference information about non-local boundary elements (r_indx). These components are dynamic, their size can be determined using Fortran 90 intrinsic operations. They are also private, meaning that the only way to manipulate the components of the mesh are by routines defined within the module. Incidentally, the remote index type r_indx (not shown) is another example of encapsulation. Objects of this type are defined so that they cannot be created outside of the module at all. A module can contain any number of derived types with various levels of protection, useful in our mesh data structure implementation strategy.

All module components are declared private, meaning that none of its components can be referenced or used outside the scope of the module. This encapsulation adds safety to the design since the internal implementation details are protected, but it is also very restrictive. Therefore, routines that must be available to module users are explicitly listed as public. This provides an interface to the module features available as the module is used in program units. Thus, the statement in the main program from Figure 1.2:

```
call mesh_create_incore( in_mesh, input_mesh_file )
```

is a legal statement since this routine is public. However the statement:

```
element_id = in_mesh%elements(10)%id
```

is illegal since the "elements" component of in_mesh is private to the derived type in the module.

Figure 1.3 Skeleton view of mesh_module components.

```
module mesh_module
use mpi_module ; use heapsort_module
implicit none
private
public :: mesh_create_incore, mesh_repartition, &
          mesh_visualize
integer, parameter :: mesh_dim=2, nodes_=3, edges_=3, neigh_=3
type element
        private
        integer :: id, nodeix(nodes_), edgeix(edges_), &
                   neighix(neigh_)
end type element
type mesh
        private
        type(node), dimension(:), pointer :: nodes
        type(edge), dimension(:), pointer :: edges
        type(element), dimension(:), pointer :: elements
        type(r_indx), dimension(:), pointer :: boundary_elements
end type mesh
contains
        subroutine mesh_create_incore(this, mesh_file)
        type (mesh), intent(inout) :: this
        character(len=*), intent(in) :: mesh_file
          ! details omitted...
        end subroutine mesh_create_incore
        ! additional member routines...
end module mesh_module
```

The mesh_module uses other modules in its definition, like the mpi_module and the heapsort_module. The mpi_module provides a Fortran 90 interface to MPI while the heapsort_module is used for efficient construction of the distributed mesh data structure. The routines defined within the contains statement, such as mesh_create_incore(), belong to the module. This means that routine interfaces, that perform type matching on arguments for correctness, are created automatically. (This is similar to function prototyping in other languages.)

1.4.2 Distributed structure organization

When the PAMR mesh data structure is constructed it is actually distributed across the processors of the parallel machine. This construction process consists of loading the mesh data, either from a single processor for parallel distribution (in_core) or from individual processors in parallel (out_of_core). A single mesh_build routine is responsible for constructing the mesh based on the data

provided. Fortran 90 routine overloading and optional arguments allow multiple interfaces to the mesh_build routine, supporting code reuse. This is helpful because the same code that builds a distributed PAMR mesh data structure from the initial description can be applied to rebuilding the data structure after refinement and mesh migration. The mesh_build routine, and its interface, is hidden from public use. Heap sorting techniques are also applied in building the hierarchical structure so that reconstruction of a distributed mesh after refinement and mesh migration can be performed from scratch, but efficiently.

The main requirement imposed on the distributed structure is that every element knows its neighbors. Local neighbors are easy to find on the current processor from the PAMR structure. Remote neighbors are known from the boundary_elements section of the mesh data structure, in combination with a neighbor indexing scheme. When an element must act on its neighbors the neighbor index structure will either have a reference to a complete description of the local neighbor element or a reference to a processor_id/global_id pairing. This pairing can be used to fetch any data required regarding the remote element neighbor. (Note that partition boundary data, such as a boundary face in three-dimensions, is replicated on processor boundaries.) One of the benefits of this scheme is that any processor can refer to a specific part of the data structure to access its complete list of non-local elements.

Figure 1.3 showed the major components of the mesh data structure, in two-dimensions. While Fortran 90 fully supports linked list structures using pointers, a common organization for PAMR codes, our system uses pointers to dynamically allocated arrays instead. There are a number of reasons why this organization is used. By using heap sorting methods during data structure construction, the array references for mesh components can be constructed very quickly. Pointers consume memory, and the memory references can become "unorganized", leading to poor cache utilization. While a pointer-based organization can be useful, we have ensured that our mesh reconstruction methods are fast enough so that the additional complexity of a pointer-based scheme can be avoided.

1.4.3 Interfacing among data structure components

The system is designed to make interfacing among components very easy. Usually, the only argument required to a PAMR public system call is the mesh itself, as indicated in Figure 1.2. There are other interfaces that exist however, such as the internal interfaces of Fortran 90 objects with MPI and the ParMeTiS parallel partitioner (Karypis et al., 1997) which were written in the C programming language.

Since Fortran 90 is backward compatible with Fortran 77 it is possible to link to MPI for interlanguage communication, assuming that the interface declarations have been defined in the mpi.h header file properly. While certain array constructs have been useful, such as array syntax and subsections, MPI does not support Fortran 90 directly so array subsections cannot be (safely) used as parameters to the library routines. Our system uses the ParMeTiS

graph partitioner to repartition the mesh for load balancing. In order to communicate with ParMeTiS our system internally converts the distributed mesh into a distributed graph. A single routine interface to C is created that passes the graph description from Fortran 90 by reference. Once the partitioning is complete, this same interface returns from C an array that describes the new partitioning to Fortran 90. This is then used in the parallel mesh migration stage to balance mesh components among the processors.

1.4.4 Interfacing among C and Fortran 90 for mesh migration

ParMeTiS only returns information on the mapping of elements to (new) processors, it can not actually migrate elements across a parallel system. Our parallel mesh migration scheme reuses the efficient mesh_build() routine to construct the new mesh from the ParMeTiS repartitioning. During this mesh_build process the element information is migrated according to this partitioning.

Figure 1.4 A main program with selected PAMR library calls.

```
subroutine mesh_repartition(this)
type (mesh), intent(inout) :: this
    ! statements omitted...
    call PARMETIS(mesh_adj, mesh_repart, nelem, nproc, iam) ! C
    call mesh_build(this, new_mesh_repart=mesh_repart)
end subroutine mesh_repartition
```

As seen in Figure 1.4, information required by the ParMeTiS partitioner is provided by calling a Fortran 90 routine that converts the mesh adjacency structure into ParMeTiS format (hidden). When this call returns from C, the private mesh_build() routine constructs the new distributed mesh from the old mesh and the new repartitioning by performing mesh migration. Fortran 90 allows optional arguments to be selected by keyword. This allows the mesh_build routine to serve multiple purposes since a keyword can be checked to determine if migration should be performed as part of the mesh construction process:

```
subroutine mesh_build(this, mesh_file, new_mesh_repart, in_core)
integer, dimension(:), intent(in), optional :: new_mesh_repart
logical, intent(in), optional :: in_core
! statements omitted...
        if (present(new_mesh_repart)) then
            ! perform mesh migration...
        end if
        ! (re)construct the mesh independent of input format...
end subroutine mesh_build
```

This is another way in which the new features of Fortran 90 add robustness to the code design. The way in which the new mesh data is presented, either from a file format or from a repartitioning, does not matter. Once the data in

organized in our private internal format the mesh can be reconstructed by code reuse.

1.4.5 Design Applications

The AMR library routines have been applied to the finite-element simulation of electromagnetic wave scattering in a waveguide filter, as well as long-wavelength infrared radiation in a quantum well infrared photodetector. Future applications may include micro-gravity experiments and other appropriate applications. This software runs on the Cray T3E, HP/Convex Exemplar, IBM SP2, and Beowulf-class pile-of-pc's running the LINUX operating system.

1.5 DO SCIENTIFIC PROGRAMS BENEFIT FROM OBJECT-ORIENTED TECHNIQUES?

Many of the benefits of object-oriented programming are probably most suited only for very large programs—typically programs larger than many scientific programmers are involved in—perhaps hundreds of thousands of lines. Nevertheless, most principles can be applied for smaller and medium scale efforts. We have applied these techniques in an experimental way on skeleton programs, but the effort addresses principles that will affect large scale development.

Object-Oriented design will not solve every problem, but it does force one to consider issues that normally might be ignored. This includes defining abstractions clearly, their relationships, and organization for extension to new problems. This increases the development time for an initial project, but hopefully reduces the effort in constructing new related projects.

One must question if the highly promoted benefits are real. Since scientific applications contain components that work together to solve complex problems, encapsulation and modularity promote good designs for these programs. The clarifies understanding, allows modifications to be introduced in an controlled manner, increases safety, and supports the work of multiple contributors. Some features, like subtyping inheritance and dynamic polymorphism are good object-oriented principles, but their general usage in scientific applications was very limited, or non-existent, in our experience. Some research has been performed in measuring the performance effects of constructs used in an object-oriented fashion in C++ and Fortran 90 (Cary et al., 1997; Norton, 1996). However, more work is needed before performance can be a deciding factor in language selection, all other factors being relatively equal. Modern applications are growing more complex, hence object-oriented techniques can clarify their organization, but this does not imply that all aspects of the paradigm are necessary.

We have experienced increased software safety, understandability by abstraction modeling, Fortran 77 level performance, and the modernization of existing programs without redevelopment in a new language. The modern features of Fortran 90 are redefining the nature of Fortran-based programming, although much interest is focused on comparing Fortran 90 and C++ for scientific pro-

gramming (Cary et al., 1997; Decyk et al., 1997b; Norton, 1996). New projects may be considering one of these languages, existing projects may reconsider their decision to adopt C++ or Fortran 90, and current projects may consider migration to the "other" language. Most of this activity grew from a realization of the new features Fortran 90 makes available as well as the continual improvement in C++ compiler technology. Scientific programming can benefit from object-oriented techniques.

1.6 CONCLUSION

The use of object-oriented concepts for Fortran 90 programming is very beneficial. The new features add clarity and safety to Fortran programming allowing computational scientists to advance their research, while preserving their investment in existing codes.

Our web site provides many additional examples of how object-oriented concepts can be modeled in Fortran 90 (Norton et al., 1996). Many concepts, like encapsulation of data and routines can be represented directly. Other features, such as inheritance and polymorphism, must be emulated with a combination of Fortran 90's existing features and user-defined constructs. (Procedures for doing this are also included at the web site.) Additionally, an evaluation of compilers is included to provide users with an impartial comparison of products from different vendors.

The Fortran 2000 standard has been defined to include explicit object-oriented features including single inheritance, polymorphic objects, parameterized derived-types, constructors, and destructors. Other features, such as interoperability with C will simplify support for advanced graphics within Fortran 2000.

Parallel programming with MPI and supercomputers is possible with Fortran 90. However, MPI does not explicitly support Fortran 90 style arrays, so structures such as array subsections cannot be passed to MPI routines. The Fortran 90 plasma PIC programs were longer than the Fortran 77 versions (but more readable), and much shorter than the C++ programs because features useful for scientific programming are not automatically available in C++. Also, the portability of the Fortran 90 parallel adaptive mesh refinement system among various machines and compilers was excellent compared to difficulties experienced with portability of C++ programs among compilers and machines.

Acknowledgments

This work was supported by a National Research Council Resident Research Associateship at the National Aeronautical and Space Administration Jet Propulsion Laboratory, California Institute of Technology. Additional support was received by the NASA Office of Space Science and the Cornell Theory Center for access to the Cray T3E and IBM SP2 respectively. We also appreciate the support of Tom A. Cwik, Viktor K. Decyk, Robert D. Ferraro, John Z. Lou, and Boleslaw K. Szymanski in this research.

References

Birdsall, C. K. and Langdon, A. B. (1991). *Plasma Physics via Computer Simulation.* The Adam Hilger Series on Plasma Physics. Adam Hilger, New York.

Cary, J. R., Shasharina, S. G., Cummings, J. C., Reynders, J. V. W., and Hinker, P. J. (1997). Comparison of C++ and Fortran 90 for Object-Oriented Scientific Programming. *Computer Physics Communications*, 105:20–36.

Decyk, V. K., Norton, C. D., and Szymanski, B. K. (1997a). Expressing Object-Oriented Concepts in Fortran 90. *ACM Fortran Forum*, 16(1):13–18.

Decyk, V. K., Norton, C. D., and Szymanski, B. K. (1997b). How to Express C++ Concepts in Fortran 90. Technical Report PPG–1569, Institute of Plasma and Fusion Research, UCLA Dept. of Physics and Astronomy, Los Angeles, CA 90095-1547.

Ellis, T. M. R., Philips, I. R., and Lahey, T. M. (1994). *Fortran 90 Programming.* Addison–Wesley, Reading, MA.

Karypis, G., Schloegel, K., and Kumar, V. (1997). ParMeTiS: Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 1.0. Technical report, Dept. of Computer Science, U. Minnesota.

Liewer, P. C. and Decyk, V. K. (1989). A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes. *J. of Computational Physics*, 85:302–322.

Norton, C. D. (1996). *Object Oriented Programming Paradigms in Scientific Computing.* PhD thesis, Renssleaer Polytechnic Institute, Troy, New York. UMI Company.

Norton, C. D., Decyk, V. K., and Szymanski, B. K. (1996). High Performance Object-Oriented Programming in Fortran 90. Internet Web Pages. http://www.cs.rpi.edu/~szymansk/oof90.html.

Norton, C. D., Decyk, V. K., and Szymanski, B. K. (1997). High Performance Object-Oriented Scientific Programming in Fortran 90. In M. Heath, et. al, editor, *Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN. CD-ROM.

Norton, C. D., Szymanski, B. K., and Decyk, V. K. (1995). Object Oriented Parallel Computation for Plasma Simulation. *Communications of the ACM*, 38(10):88–100.